Product line engineering: Product derivation

Laurent Monestel, Tewfik Ziadi, Jean-Marc Jézéquel

Product Line Engineering : Product Derivation

Laurent Monestel, Tewfik Ziadi and Jean-Marc Jézéquel

IRISA (INRIA & University of Rennes), Campus de Beaulieu
Contact : laurent.monestel@irisa.fr; +33 2 99 84 74 87

Abstract. Handling the various derivations of a product can be a daunting (and costly) task. To tackle this problem, we propose a method based on the use of creational design patterns to uncouple the variations (reified as language-level objects) from the selection process. This makes it possible to automatically derive a given product from the set of all possible ones, and to specialize its code accordingly. The contribution of this paper is to propose architectural constraints for Product Line expressed in UML as meta-level OCL constraints, providing a set of patterns for modeling variability issues of a Product Line Architecture in the context of the OMG's Model Driven Architecture, and explicit the use of OCL2 transformations combined together using the UMLAUT framework to automate the derivation process.

Software Product Line with UML

Software Product Line (SPL) captures "commonality" and "variability" between a set of software products sharing a common, managed set of features that satisfy the specific needs of a particular market segment. Commonality designates the elements that are common to all products while variability designates the elements that may vary from a product to another one. Software Product Line Architecture (SPLA) is a generic software architecture that applies to a set of products and from which the software architecture of each product can be derived [1]. The main goal of a SPL is to model and implement a set of reusable assets that will be used to derive specific software products.

The main challenge in the context of SPL is to model and implement variability. While Unified Modeling Language (UML) [2] does not explicitly support SPLA, in this section, we intend to show how we can use UML and its extension mechanisms to model static aspects of a Product Line and especially variability.

Object Oriented modeling of variants: Class diagram

- *Abstraction*. Using an object-oriented analysis and design approach, it is natural to model the commonalities between the variants of a variation point in an abstract class (or interface), and expressing the differences in concrete subclasses (each variant implements the interface in its own way).
- *Parameterization*: Classes can be defined as generic assets with a set of parameters, each product bind these parameters in a specific way. We use UML class template to specify parameterization classes.

- *Optionality*: The Product Line model include all the elements associated to all products, so in such products some of these elements called optional will be omitted. To show optionality information we define an ad hoc stereotype «optional».

Mercure Case Study

As a case study for evaluating the interest of our approach, we consider the Mercure project, which is an SMDS (Switched Multi-Megabits Data Service) server whose design and implementation have been described in [3; 4].It can abstractly be described as a communication software delivering, forwarding and relaying “messages” from/to a set of network interfaces connected onto an heterogeneous distributed system.

Mercure must handle variants for five variation points: any number of network interface boards, specialized processors, levels of functionality, user interface and support for languages.

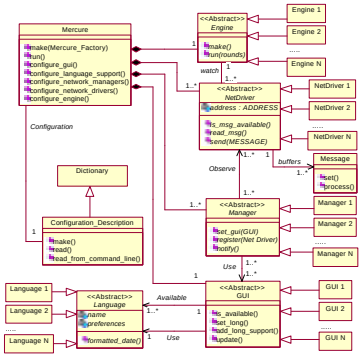

Fig. 1. The initial Mercure Product Line model

The use of OCL constraints in a Product Line Architecture

Bass et al [5] consider that constraints are a part of SPLA. Constraints define the relationship between software architecture elements. Some work such as [6] use UML stereotypes to show the dependencies between classes.

The Object Constraints Language (OCL) [7] allows us to attach constraints to UML models. These constraints can be defined at meta-model level as well as model level. In the context of SPLA, we have identified two types of constraints, generic constraints applying to any SPLA, and specific constraints applying to a specific Product Line.

Examples of constraints applying to any Product Line Architecture

- *Inheritance constraint.* Optional classes in Product Line Architecture can be omitted in certain products then, if a non-optional class inherit from an optional one, there’s an incoherence in the model.

```
context Generalization
  inv self.parent.isStereotyped[1](“optional”) implies
self.chield.isStereotyped(“optional”)
```

- *Dependency constraint.* A dependency in UML specify a requirement relationship between two or more elements, if a non-optional element is depending on an optional one, there’s an incoherence in the model.

```
context Dependency
  inv self.supplier->exists( S:ModelElement |
S.isStereotyped(“optional”) )
implies self.client->forAll( C:ModelElement |
C.isStereotyped(“optional”) )
```

Examples of constraints for a specific Product Line Architecture

- *Presence constraint.* To express in a specific SPLA that the presence of the optional class C1 requires the presence of C2, we add the following OCL meta-model constraint.

```
context Namespace
  inv presenceClass(self, C1) implies
presenceClass(self, C2)
```

- *Mutual Exclusion constraint.* To express in a specific SPLA that two optional classes cannot be present in the same Product, we add the following OCL meta-model constraint.

```
context Namespace
  inv (presenceClass(self, C1) implies not
presenceClass(self, C2)) and (presenceClass(self, C2)
implies not presenceClass(self, C1))
```

From Product Line to Product

Product configuration in a Product Line

Once we have analyzed our Product Line and produced the corresponding Model, we still need to handle the various derivations of a Product (Decision Model [1]).

Creational Design Patterns as proposed in [8] can help make this process easily customizable by uncoupling the system from how its constituent objects get created, composed and represented. In our simple case, we use an Abstract Factory, to define an interface for creating variants of Mercure’s five variation points.

Obtaining an actual Product of the Mercure Product Line then consists in implementing the relevant concrete factory. Once we have designed each product with a concrete factory, the selection of such a factory allows the designer to specify the Product he want.

[1] *isStereotyped*() and *presenceClass*() are our own high-level OCL operations.

Product derivation using OCL2

At this stage, we have precisely defined the Product Line , now we have to tackle with the automation of the derivation process. To achieve this we propose the use of the UMLAUT framework [9] combined to an OCL2 transformation.

UMLAUT is a framework for building tools dedicated to the manipulation of models described using the UML. A specific use is to apply a model transformation to an UML model, automating the derivation process then consists in writing the relevant model transformation.

This transformation retrieves the useful model elements thanks to the selected concrete factory and then builds a specialized UML model corresponding to the selected Product. The challenge of such a model manipulation is to be able to transform the model accessing its meta-level and ensuring the integrity of the model by preserving all of his constraints. The interest of using OCL2 is there, among being part of UML2.0 standard, this language is an extension of OCL and Action Semantics, so by definition, it permits us to describe the process at the meta-level and to interpret (to preserve) OCL constraints.

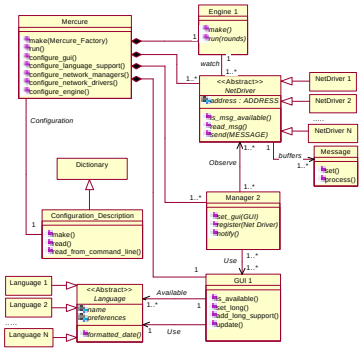

Fig. 2. A derived Product model

References

1. Elkaim, W.-E 2001. System family architecture glossary. Technical report, ESAPS Project.
2. OMG. UML specification.
3. Jézéquel, J.-M. 1996. Object Oriented Software Engineering with Eiffel. Addison-Wesley. ISBN 1-201-63381-7.
4. Jézéquel, J.-M. 1998. Object-orented design of real-time telecom systems. In IEEE International Symposium on Object-oriented Real-time distributed Computing, ISORC'98, Kyoto, Japan (April 1998).
5. Clements, P., Bass, L., and Kazman, R. 1998. Software Architecture in practices. Addison-Wesley.
6. Clauß, M., 2001 Modeling variability with UML. In GCSE 2001 Young researchers Workshop.
7. Warmer, J., and Kleppe, A. 1998. The Object Constraint Language – Precise Modeling with UML. Object Technology Series. Addison-Wesley.
8. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.

9. Ho, W.-M., Jézéquel, J.-M., Le Guennec, A., and Pennaneac'h, F. UMLAUT : an extensible UML transformation framework. In Proc. Automated Software Engineering, ASE'99, Florida, October 1999.